**Functional Reactive Programming**

What is Functional Reactive Programming (FRP)?
- Paradigm for reactive programming in a functional setting.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.

**FRP applications**

Some domains where FRP has been used:
- Graphical Animation (Fran: Elliott, Hudak)
- Robotics (Frob: Peterson, Hager, Hudak, Elliott, Pembeci, Nilsson)
- Vision (FVision: Peterson, Hudak, Reid, Hager)
- GUIs (Fruit: Courtney)
- Hybrid modeling (Nilsson, Hudak, Peterson)

**Key FRP features**

- First class reactive components.
- Synchronous: all system parts operate in synchrony.
- Support for hybrid (mixed continuous and discrete time) systems.
- Allows dynamic system structure.

---

**Reactive programming**

**Reactive systems:**
- Input arrives *incrementally* while system is running.
- Output is generated in response to input in an interleaved and *timely* fashion.

Contrast *transformational systems*.
The notions of
- time
- time-varying values, or *signals* are inherent and central for reactive systems.

**Yampa**

What is Yampa?
- The most recent Yale FRP implementation.
- People: Antony Courtney, Paul Hudak, Henrik Nilsson, John Peterson

**Related languages and paradigms**

FRP related to:
- Synchronous languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink, Modelica.
Yampa?

Yampa is a river with long calmly flowing sections and abrupt whitewater transitions in between.

A good metaphor for hybrid systems!

Signal functions in Yampa

- **Signal functions** are first class entities. Intuition: \( \text{SF} \alpha \beta \equiv \text{Signal} \alpha \rightarrow \text{Signal} \beta \)
- **Signals** are not first class entities: they only exist indirectly through signal functions.

Example: Robotics (1)

[PPDP'02, with Izzet Pembeci and Greg Hager, Johns Hopkins University]

Hardware setup:

Example: Video tracker

Video trackers are typically stateful signal functions:

Example: Robotics (2)

Software architecture:

Example: Robotics (3)
In Yampa, systems are described by combining signal functions (forming new signal functions). For example, serial composition:

A combinator can be defined that captures this idea:

\[
(++) : SF a b \rightarrow SF b c \rightarrow SF a c
\]

But systems can be complex:

How many and what combinators do we need to be able to describe arbitrary systems?

John Hughes’ arrow framework:
- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
- Related to monads, since arrows are computations, but more general.
- Provides a minimal set of “wiring” combinators.

What is an arrow? (1)
- A type constructor \( a \) of arity two.
- Three operators:
  - lifting: \( \text{arr} :: (b\rightarrow c) \rightarrow a b c \)
  - composition: \( (++) :: a b c \rightarrow a c d \rightarrow a b d \)
  - widening: \( \text{first} :: a b c \rightarrow a (b,d) (c,d) \)
- A set of algebraic laws that must hold.

What is an arrow? (2)

These diagrams convey the general idea:

The Arrow class
In Haskell, a type class is used to capture these ideas (except for the laws):

Functions are arrows (1)
Functions are a simple example of arrows. The arrow type constructor is just \((\rightarrow)\) in that case.

Exercise 1: Suggest suitable definitions of
- \(\text{arr}\)
- \((++)\)
- \(\text{first}\)
for this case!

(We have not looked at what the laws are yet, but they are “natural”.)

Functions are arrows (2)
Solution:
- \(\text{arr} = \text{id}\)
  To see this, recall
  \(\text{id} :: t \rightarrow t\)
  \(\text{arr} :: (b\rightarrow c) \rightarrow a b c\)
  Instantiate with
  \(a = (\rightarrow)\)
  \(t = b\rightarrow c = (\rightarrow) b c\)

Functions are arrows (3)
- \(f ++ g = \lambda a \rightarrow g (f a)\) or
- \(f ++ g = g \cdot f\) or even
- \((++) = \text{flip} (.).\)
- \(\text{first} f = \lambda (b,d) \rightarrow (f b,d)\)
Functions are arrows (4)

Arrow instance declaration for functions:

```haskell
instance Arrow (->) where
  arr    = id
  (<<<>) = flip (.)
  first f = \(b,d) -> (f b,d)
```

Arrow laws

```haskell
(f <<< g) <<< h = f <<< (g <<< h)
arr (f <<< g) = arr f <<< arr g
arr id <<< f = f
  f = f <<< arr id
first (arr f) = arr (first f)
first (f <<< g) = first f <<< first g
```

Exercise 2: Draw diagrams illustrating the first and last law!

The loop combinator (1)

Another important operator is loop: a fixed-point operator used to express recursive arrows or feedback:

```
loop f
```

The loop combinator (2)

Not all arrow instances support `loop`. It is thus a method of a separate class:

```haskell
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
```

Remarkably, the four combinators `arr`, `<<<>`, `first`, and `loop` are sufficient to express any conceivable wiring!

Some more arrow combinators (1)

```haskell
second :: Arrow a => a b c -> a (d,b) (d,c)
(<<<>) :: Arrow a => a b c -> a (d,e) -> a b (c,e)
(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)
```

Some more arrow combinators (2)

As diagrams:

```
second f
f <<< g
```

Some more arrow combinators (3)

Exercise 3: Describe the following circuit using arrow combinators:

```
a1 a2 a3
```

a1, a2, a3 :: A Double Double

Exercise 4: The combinators `second`, `(<<<>)`, and `(&&&)` are not primitive, but defined in terms of `arr`, `<<<>`, and `first`. Suggest suitable definitions!

Reading (1)


Reading (2)